Asynchronous Memory Machine Models with Barrier Synchronization

Koji Nakano
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan
Email: nakano@cs.hiroshima-u.ac.jp

***Abstract*—The Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) are theoretical parallel computing models that capture the essence of the shared memory and the global memory of GPUs. It was assumed that warps (i.e. groups of threads) on the DMM and the UMM work synchronously in the round-robin manner. However, warps work asynchronously in the actual GPUs, in the sense that warps may be randomly (or arbitrarily) dispatched for execution. The first contribution of this paper is to introduce an asynchronous version of the DMM and the UMM, in which warps are arbitrarily dispatched. Instead, we assume that threads can execute the "syncthreads" instruction for barrier synchronization. Since the barrier synchronization operation is costly, we should evaluate and minimize the number of barrier synchronization operations performed by parallel algorithms. The second contribution of this paper is to show a parallel algorithm to compute the sum of n numbers in optimal computing time and few barrier synchronization steps. Our parallel algorithm computes the sum of n numbers in $O(\frac{n}{w}+l\log n)$ time units and $O(\log \frac{l}{w}+\log\log w)$ barrier synchronization steps using wl threads both on the asynchronous DMM and on the asynchronous UMM with width w and latency l. We also prove that the computing time is optimal because it matches the theoretical lower bound. Quite surprisingly, the number of barrier synchronization steps and the number of threads are independent of n. Even if the input size n is quite large, our parallel algorithm computes the sum in optimal time units and a fixed number of syncthreads using a fixed number of threads.**

***Keywords*-parallel computing models, parallel algorithms, contiguous memory access, asynchronous models, GPU, CUDA**

I. Introduction

The research of parallel algorithms has a long history of more than 40 years. Sequential algorithms have been developed mostly on the Random Access Machine (RAM) [1]. In contrast, since there are a variety of connection methods and patterns between processors and memories, many parallel computing models have been presented and many parallel algorithmic techniques have been shown on them. The most well-studied parallel computing model is the Parallel Random Access Machine (PRAM) [2], [3], [4], which consists of processors and a shared memory. Each processor on the PRAM can access any address of the shared memory in a time unit. The PRAM is a good parallel computing model in the sense that parallelism of each problem can be revealed by the performance of parallel algorithms on the PRAM. However, since the PRAM requires a shared memory that can be accessed by all processors at the same time, it is not feasible.

The GPU (Graphical Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [5], [6], [7], [8], [9]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [5], [10], [11]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [12], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [13], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [12]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. The global memory is implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [7], [13], [14]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed sequentially. Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

In our previous paper [15], we have introduced two models, *the Discrete Memory Machine (DMM)* and *the*

Unified Memory Machine (UMM), which reflect the essential features of the shared memory and the global memory of NVIDIA GPUs. The outline of the architectures of the DMM and the UMM is illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* is connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [1], which can execute one of the fundamental operations in a time unit. We do not discuss the architecture of the sea of threads in this paper, but we can imagine that it consists of a set of multi-core processors which can execute many threads in parallel and/or in time-sharing manner. Threads are executed in SIMD [16] fashion, and the processors run on the same program and work on the different data.

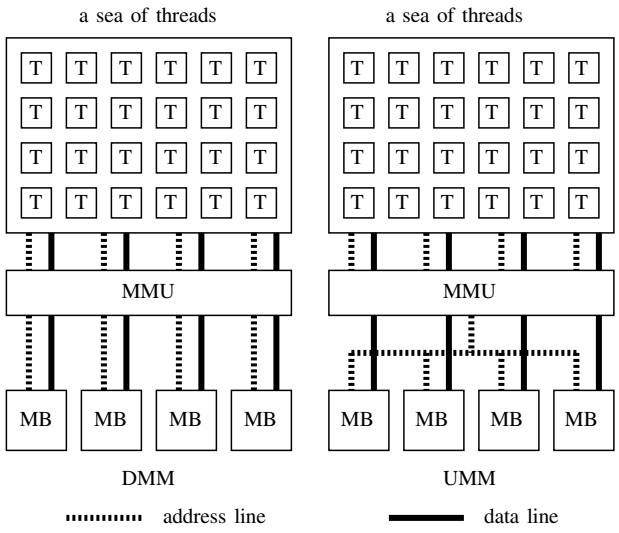

Figure 1. The architectures of the DMM and the UMM

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM.

The performance of algorithms on the PRAM is usually evaluated using two parameters: the size n of the input and the number p of processors. For example, it is well known that the sum of n numbers can be computed in $O(\frac{n}{p}+\log p)$ time on the PRAM [2]. We will use four parameters, the size n of the input, the number p of threads, the width w and the latency l of the memory when we evaluate the performance of algorithms on the DMM and on the UMM. The width w is the number of memory banks and the latency l is the number of time units to complete the memory access. Hence, the performance of algorithms on the DMM and the UMM is evaluated as a function of n (the size of a problem), p (the number of threads), w (the width of a memory), and l (the latency of a memory). Further, r (the number of local registers used by each thread) may be additionally used.

Note that width w and latency l depend on the architecture. They are fixed values and cannot be changed. On the other hand, the number p of threads can be changed. Users can choose optimal value of p to get the best performance. Thus, the computing time of algorithms on the DMM and the UMM can be evaluated without using p. For example, in our previous paper [17], we have shown that the prefix-sums of n numbers can be computed in $O(\frac{n}{w}+\frac{nl}{p}+l\log n)$ time units on the DMM and the UMM. To get the best performance, w should choose $p = wl$. If this is the case, the prefix-sums can be computed in $O(\frac{n}{w}+l\log n)$ time units.

Suppose that we use p threads $T(0), T(1), \ldots, T(p-1)$. Threads on the DMM and the UMM are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. Let $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ denote the $\frac{p}{w}$ groups. In our previous paper, it is assumed that threads the DMM and the UMM works *synchronously* in the sense that warps are activated for memory access from $W(0)$ to $W(\frac{p}{w}-1)$ in turn by the round-robin manner. The first contribution of this paper is to extend memory machine models presented in our previous paper [15] for more realistic parallel computing models. More specifically, we assume that threads works *asynchronously* in the sense that warps are dispatched for memory access arbitrarily. The scheduler arbitrarily selects one of the warps in which at least one thread tries to access the memory, and dispatches it for memory access. Instead, we assume that threads can execute an instruction *syncthreads* for the purpose of barrier synchronization. In NVIDIA GPUs, `__syncthreads()` instruction is supported for threads in a block, which takes $\frac{p}{16}$ clock cycles [12]. Also, for the purpose of synchronization of threads in multiple blocks we need to separate algorithm into different kernel calls [12]. Hence, barrier synchronization is costly. In this paper, when we evaluate the performance of parallel algorithm on the asynchronous DMM and the asynchronous UMM, we also evaluate the number of *syncthreads* operations performed. Note that, parallel algorithms on the asynchronous versions of the DMM and the UMM must work correctly for any worst choice of warps by a malicious scheduler. Also, the performance including the computing time must be evaluated for the case of worst choice of warps.

The second contribution of this paper is to show efficient

summing algorithm on the asynchronous version of the DMM and the UMM with width w and latency l. We first show that a simple algorithm shown in [17] can compute the sum of n numbers in $O(\frac{n}{w}+l\log n)$ time units and $O(\log \frac{n}{w})$ barrier synchronization steps (Algorithm Simple). We then go on to prove that $\Omega(\frac{n}{w}+l\log n)$ time units are necessary to compute the sum of n numbers. Thus, Algorithm Simple is time optimal. We also show that the sum of n numbers can be computed in $O(\frac{n}{w}l+l\log n)$ time units and 0 barrier synchronization step (Algorithm One-Warp). Although this algorithm does not perform barrier synchronization, it is not time optimal and has large overhead of factor l. Next, we will show that a parallel algorithm based on a $2w$-ary tree can compute the sum of n numbers in $O(\frac{n}{w}\log w+l\log n)$ time units and $O(\frac{\log n}{\log w})$ barrier synchronization steps (Algorithm Tree). By combining Algorithm Simple and Algorithm Tree, we show that the sum of n numbers can be computed in $O(\frac{n}{w}+l\log n)$ time units and $O(\frac{\log n}{\log w}+\log\log w)$ barrier synchronization steps (Algorithm Simple-Tree). Clearly, Algorithm Sump-Tree is time optimal. Finally, we will show that the barrier synchronization steps can be reduced to $O(\frac{\log l}{\log w}+\log\log w)$ (Algorithm Hybrid). Quite surprisingly, the number of barrier synchronization steps and the number of threads of Algorithm Hybrid are independent of n. Even if the input size n is quite large, our parallel algorithm computes the sum in optimal time units and a fixed number of syncthreads using a fixed number of threads. Table I summarizes our summing algorithms presented in this paper.

This paper is organized as follows. Section II defines the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) introduced in our previous paper [15] and define the asynchronous version of the DMM and the UMM. In Section III, we evaluate the computing time of the contiguous memory access to the memory of the asynchronous DMM and the asynchronous UMM. The contiguous memory access is a key ingredient of parallel algorithm development on the memory machine models. Using the contiguous access, we show that Algorithm Simple can compute the sum of n numbers in $O(\frac{n}{w}+l\log n)$ time units and $O(\frac{\log n}{\log w})$ barrier synchronization steps in Section IV. We also discuss the lower bound of the time complexity and show two lower bounds, $\Omega(\frac{n}{w})$-time bandwidth limitation and $\Omega(l\log n)$-time reduction limitation. Section V shows Algorithm One-Warp that computes the sum of n numbers in $O(\frac{n}{w}l+l\log w)$ time units and 0 synchronization steps. In Section VI shows an tree-based summing algorithm Algorithm Tree that computes the sum of n numbers in $O(\frac{n}{w}\log w+l\log w)$ time units and $O(\frac{\log n}{\log w})$ barrier synchronization steps. Finally, SectionVII shows time-optimal summing algorithm. Algorithm Simple-Tree, which is a combination of Algorithm Simple and Algorithm Tree, uses $O(\frac{\log n}{\log w}+\log\log w)$ barrier synchronization steps. By an appropriate precomputation, we show that the barrier synchronization steps can be reduced to $O(\frac{\log l}{\log w}+\log\log w)$. Section VIII offers concluding remarks.

II. Parallel Memory Machines: DMM and UMM

The main purpose of this section is to define the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM). introduced in our previous paper [15], [17].

We first define *the Discrete Memory Machine (DMM)* of width w and latency l, Let $m[i]$ $(i \geq 0)$ denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j+w], m[j+2w], m[j+3w], \ldots\}$ $(0 \leq j \leq w-1)$ denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the $(i \bmod w)$-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells at the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, it takes $k+l-1$ time units to complete k access requests to a particular bank.

We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i\cdot w), T(i\cdot w+1), \ldots, T((i+1)\cdot w-1)\}$ $(0 \leq i \leq \frac{p}{w}-1)$. Warps are dispatched for memory access in turn and w threads in a warp try to access the memory at the same time. We define two assumptions *synchronous manner* and *asynchronous manner* in terms of dispatching of warps. In the synchronous manner, $W(0), W(1), \ldots, W(w-1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. More specifically, suppose that every thread executes $\mathcal{T}$ instructions. In the synchronous manner, warps work equally as follows:

[Synchronous Model]
for $t \leftarrow 0$ to $\mathcal{T}$ do
  for $i \leftarrow 0$ to $\frac{p}{w}-1$ do
    Every thread in $W(i)$ executes an instruction.

On the other hand, in asynchronous operations, one of the warps is dispatched and executed as follows:

[Asynchronous Model]
for $t \leftarrow 0$ to $\frac{\mathcal{T}p}{w}-1$ do
  Arbitrarily select a warp $W(i)$ to be executed.
  Each thread in $W(i)$ executes an instruction.

Note that, in asynchronous manner, if all threads in a warp $W(i)$ have no instruction to be executed, such warp $W(i)$ is not selected. For example, if threads in $W(i)$ have just sent memory access requests and they are waiting for completion of memory access, $W(i)$ is not selected. Such warp $W(i)$ will be selected after the completion of memory access.

We also assume that, for the purpose of barrier synchronization, all threads can execute the *syncthreads* instruction. Suppose that at least one of the p threads executes *syncthreads*. After that, all threads that have executed

Table I
PERFORMANCE OF PARALLEL ALGORITHM FOR COMPUTING THE SUM

algorithms	time units	threads	syncthreads	time optimality
Simple	$O(\frac{n}{w}+l\log n)$	$\frac{n}{2}$	$O(\log \frac{n}{w})$	optimal
One-Warp	$O(\frac{n}{w}l+l\log n)$	w	0	overhead of factor l
Tree	$O(\frac{n}{w}\log w+l\log n)$	n	$O(\frac{\log n}{\log w})$	overhead of factor $\log w$
Simple-Tree	$O(\frac{n}{w}+l\log n)$	$\frac{n}{2}$	$O(\frac{\log n}{\log w}+\log\log w)$	optimal
Hybrid	$O(\frac{n}{w}+l\log n)$	wl	$O(\frac{\log l}{\log w}+\log\log w)$	optimal

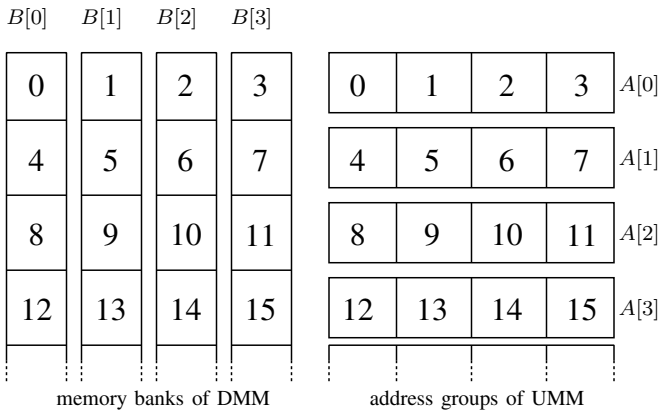

Figure 2. Banks and address groups for $w = 4$

syncthreads have been blocked until all threads execute *syncthreads*. Once all threads execute *syncthreads*, they restart executing instructions.

We assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread send a memory access request, it must wait l time units to send a new memory access request.

For the reader's benefit, let us evaluate the time for memory access using Figure 3 on the DMM for $p = 8$, $w = 4$, and $l = 3$. In the figure, $p = 8$ threads are partitioned into $\frac{p}{w} = 2$ warps $W(0) = \{T(0), T(1), T(2), T(3)\}$ and $W(1) = \{T(4), T(5), T(6), T(7)\}$. As illustrated in the figure, 4 threads in $W(0)$ try to access $m[0], m[1], m[6]$, and $m[10]$, and those in $W(1)$ try to access $m[8], m[9], m[14]$, and $m[15]$. The time for the memory access are evaluated under the assumption that memory access are processed by imaginary l pipeline stages with w registers each as illustrated in the figure. Each pipeline register in the first stage receives memory access request from threads in an activated warp. Each i-th ($0 \leq i \leq w-1$) pipeline register receives the request to the i-th memory bank. In each time unit, a memory request in a pipeline register is moved to the next one. We assume that the memory access completes when the request reaches the last pipeline register.

Note that, the architecture of pipeline registers illustrated in Figure 3 are imaginary, and it is used only for evaluating the computing time. The actual architecture should involves a multistage interconnection network [18], [19] or sorting network [20], [21], to route memory access requests.

Let us evaluate the time for memory access on the DMM. First, access request for $m[0], m[1], m[6]$ are sent to the first stage. Since $m[6]$ and $m[10]$ are at the same bank $B[2]$, their memory requests cannot be sent to the first stage at the same time. Next, the $m[10]$ is sent to the first stage. After that, memory access requests for $m[8], m[9], m[14], m[15]$ are sent at the same time, because they are in different memory banks. Finally, after $l-1 = 2$ time units, these memory requests are processed. Hence, the DMM takes 5 time units to complete the memory access.

We next define *the Unified Memory Machine* (*UMM*) of width w as follows. Let $A[j] = \{m[j \cdot w], m[j \cdot w + 1], \ldots, m[(j+1) \cdot w - 1]\}$ denote the j-th address group. We assume that memory cells at the same address group are processed at the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp accesses the memory in turn.

Again, let us evaluate the time for memory access using Figure 3 on the UMM for $p = 8$, $w = 4$, and $l = 3$. The memory access requests by $W(0)$ are in three address

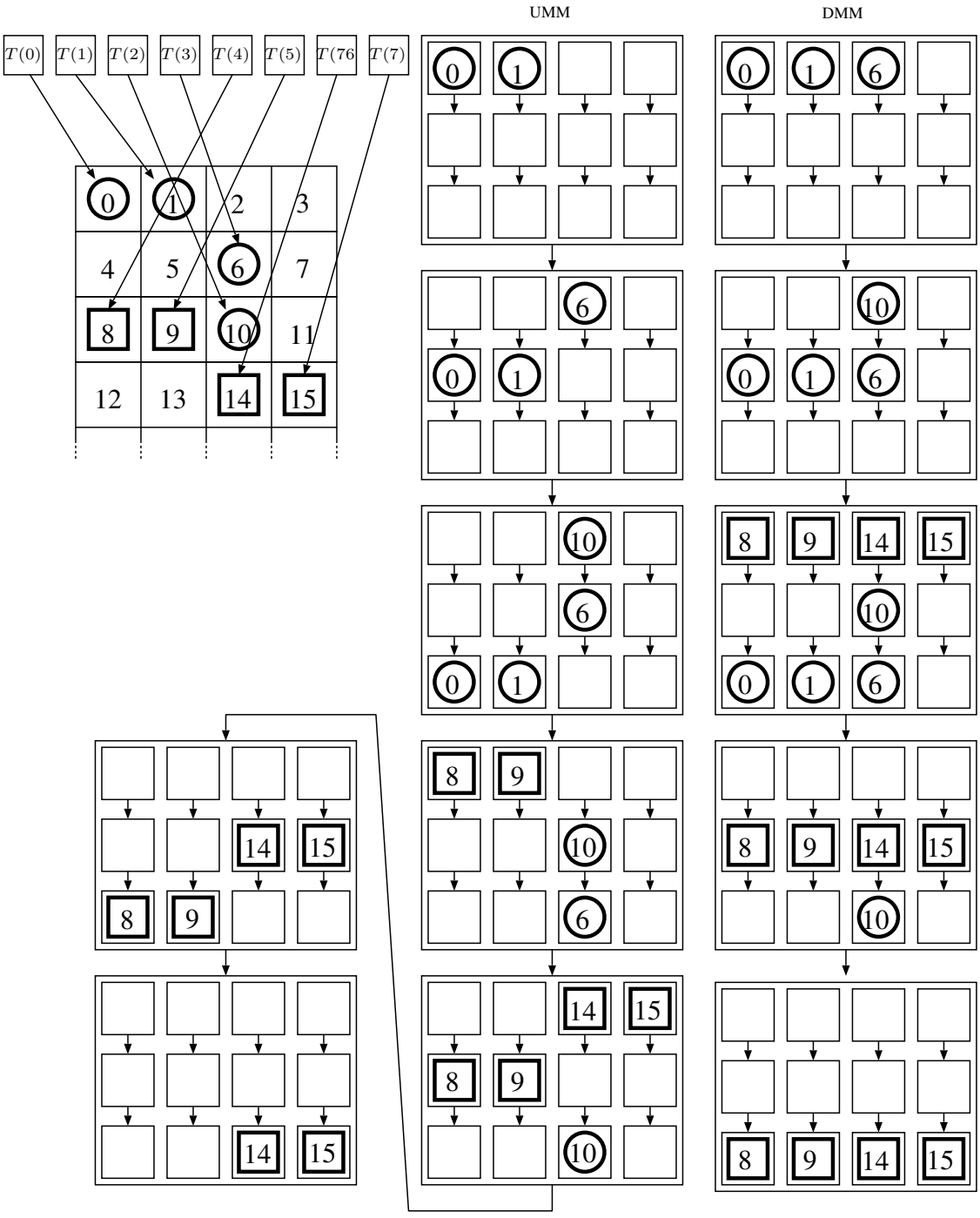

Figure 3. An example of memory access

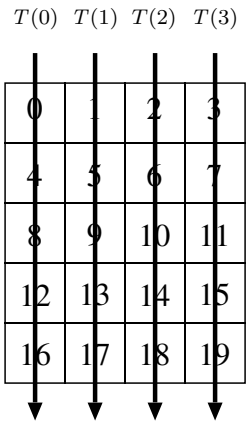

Figure 4. Contiguous memory access for $n = 20$ and $p = 4$.

groups. Thus, three time units are necessary to send them to the first stage. Next, two time units are necessary to send memory access requests by $W(1)$, because they are in two address groups. After that, it takes $l - 1 = 2$ time units to process the memory access requests. Hence, totally $3 + 2 + 2 = 7$ time units are necessary to complete all memory access.

III. Contiguous Memory Access

The main purpose of this section is to show the contiguous memory access on the asynchronous DMM and the asynchronous UMM. The evaluation of the computing time for the contiguous access on the synchronous DMM and the synchronous UMM is not difficult [15], [17]. However, that for the asynchronous version is more complicated. This section shows the computing time on the asynchronous DMM and the synchronous UMM is the same as that on the synchronous version.

Suppose that an array a of size n ($\geq p$) is given. We use p threads to access all of n memory cells in a such that each thread accesses $\frac{n}{p}$ memory cells. Note that "accessing" can be "reading from" or "writing in." Let $a[i]$ $(0 \leq i \leq n-1)$ denote the i-th memory cells in a. We can consider that a is a 2-dimensional array of size $\frac{n}{p} \times p$ ($\frac{n}{p}$ rows and p columns). Each $a[i][j]$ $(0 \leq i \leq \frac{n}{p} - 1, 0 \leq j \leq p-1)$ corresponds to $a[i \cdot p + j]$. The contiguous memory access can be performed as follows:

[Contiguous memory access]
for $i \leftarrow 0$ to $p-1$ do in parallel
 for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
  $T(i)$ accesses $a[t][i]$.

Figure 4 illustrates the contiguous memory access for $n = 20$ and $p = 4$.

Let evaluate the computing time. Each warp $W(j)$ $(0 \leq j \leq \frac{p}{w} - 1)$ with w threads access to w memory cells $a[t][j \cdot w], a[t][j \cdot w + 1], \ldots, a[t][(j+1) \cdot w - 1]$ for each t $(0 \leq t \leq \frac{n}{p})$. In other words, each warp $W(i)$ repeatedly access w memory cells at the same address group $\frac{p}{w}$ times. We will evaluate the computing time for the following two cases:

Case 1: $\frac{p}{w} < l$ First, one of the warps is randomly dispatched and sends memory access requests. After a warp sends requests, it will not be selected at least l time units. Thus, all of the $\frac{p}{w}$ warps are dispatched in the first $\frac{p}{w}$ time units. Each warp takes l time units to complete the memory access, Thus, the second memory access is started at time l. Figure 5 illustrates how contiguous memory access is performed when $\frac{p}{w} < l$. Contiguous memory access requests by $\frac{p}{w}$ warps are repeatedly sent $\frac{n}{p}$ times. Thus, it takes $\frac{p}{w} + l \cdot \frac{n}{p} = O(\frac{nl}{p})$ time units for the contiguous memory access.

Case 2: $\frac{p}{w} \geq l$ Each of the $\frac{p}{w}$ warps sends memory access requests $\frac{n}{p}$ times. Hence, totally they send memory access requests $\frac{n}{p} \cdot \frac{p}{w} = O(\frac{n}{w})$ times. Clearly, if at least l warps have not completed memory access, they can send memory access request continuously. On the other hand, if no warp send memory access request in a time unit, then less than l warps still have memory access requests to be sent. Hence each warp in less than l such warps can send memory access requests at least once in l time units. Since each warps send memory access $\frac{n}{p}$ times, it takes $l \cdot \frac{n}{p} = O(\frac{nl}{p})$ time units for less than l such warps to complete the memory access requests. Therefore, the contiguous memory access can be completed in $O(\frac{n}{w} + \frac{nl}{p})$ time units. Thus, we have,

Lemma 1: The contiguous access to an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p})$ time units with 0 barrier synchronization step using p threads on the UMM and the DMM with width w and latency l.

IV. A simple summing algorithm and the time lower bound

The main purpose of this section is to show a simple parallel algorithm for computing the sum on the memory machine models. The summing algorithm presented in this section is the essentially same as one presented in [17] on the synchronous DMM and the synchronous UMM.

Let a be an array of $n = 2^m$ numbers. Let us show an algorithm to compute the sum $a[0]+a[1]+\cdots+a[n-1]$. The algorithm uses a well-known parallel computing technique which repeatedly computes the sums of pairs. We implement this technique to perform contiguous memory access using $\frac{n}{2}$ threads. The details are spelled out as follows:

[Algorithm Simple]
for $t \leftarrow m-1$ down to 0 do
 begin
  for $i \leftarrow 0$ to $2^t - 1$ do in parallel
   T(i) performs $a[i] \leftarrow a[i] + a[i + 2^t]$
  if($2^t > w$) *syncthreads*
 end

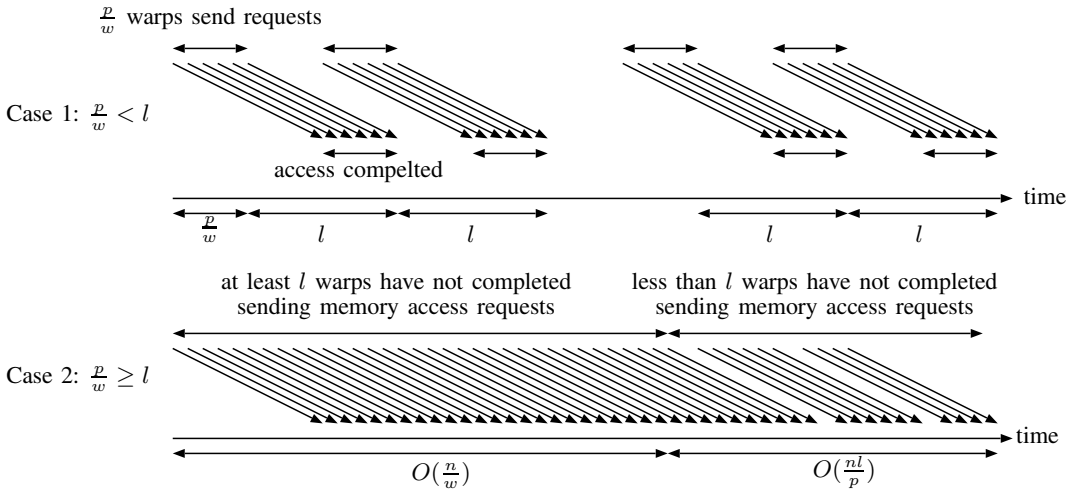

Figure 5. Contiguous memory access when $\frac{p}{w} < l$

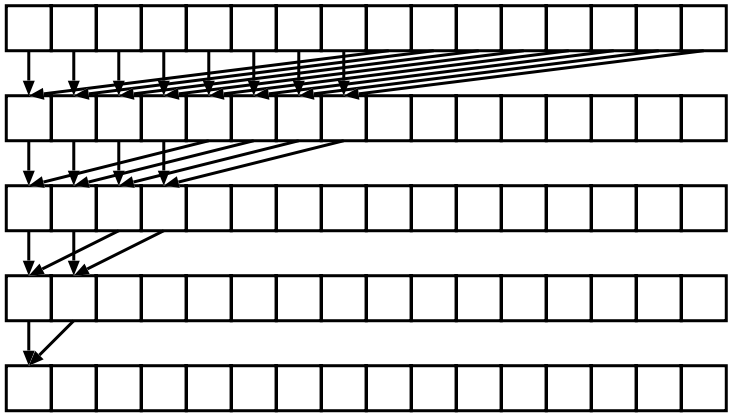

Figure 6. Illustrating the summing algorithm for n numbers

Figure 6 illustrates how the sums of pairs are computed. From the figure, it should be clear that this algorithm compute the sum correctly.

Let us evaluate the computing time. For each t ($0 \leq t \leq m-1$), 2^t operations "$a[i] \leftarrow a[i] + a[i+2^t]$" are performed. These operation involve the following memory access operations:

- reading from $a[0], a[1], \ldots, a[2^t - 1]$,
- reading from $a[2^t], a[2^t + 1], \ldots, a[2 \cdot 2^t - 1]$, and
- writing in $a[0], a[1], \ldots, a[2^t - 1]$,

Since these memory access operations are contiguous, they can be done in $O(\frac{2^t}{w} + \frac{2^t l}{2^t}) = O(\frac{2^t}{w} + l)$ time using 2^t threads both on the DMM and on the UMM with width w and latency l from Lemma 1. Thus, the total computing time is

$$\sum_{t=0}^{m-1} O(\frac{2^t}{w} + l) \quad = \quad O(\frac{2^m}{w} + lm)$$

$$= \quad O(\frac{n}{w} + l \log n).$$

Barrier synchronization *syncthreads* is executed $m - \log w = O(\log \frac{n}{w})$ times. Thus, we have,

Lemma 2: Algorithm Simple computes the sum of n numbers in $O(\frac{n}{w} + l \log n)$ time units and $O(\log \frac{n}{w})$ barrier synchronization steps using $\frac{n}{2}$ threads on the DMM and on the UMM with width w and latency l.

Note that if $n \leq w$, then only one warp is used and thus *syncthreads* is not necessary.

Let us discuss the lower bound of the time necessary to compute the sum on the DMM and the UMM to show that our parallel summing algorithm for Lemma 2 is optimal. We will show two lower bounds, $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(l \log n)$-time reduction limitation.

Since the width of the memory is w, at most w numbers in the memory can be read in a time unit. Clearly, all of the n numbers must be read to compute the sum. Hence, $\Omega(\frac{n}{w})$

time units are necessary to compute the sum. We call the $\Omega(\frac{n}{w})$-time lower bound *the bandwidth limitation*.

Each thread can perform a binary operation such as addition in a time unit. If at least one of the two operands of a binary operation is stored in the shared memory, it takes at least l time units to obtain the resulting value. Clearly, addition operation must be performed $n-1$ times to compute the sum of n numbers. The computation of the sum using addition is represented using a binary tree with n leaves and $n-1$ internal nodes. The root of the binary tree corresponds to the sum. From basic graph theory, there exists a path from the root to a leaf, which has at least $\log n$ internal nodes. The addition corresponds to each internal node takes l time units. Thus, it takes at least $\Omega(l \log n)$ time to compute the sum, regardless of the number p of threads. We call the $\Omega(l \log n)$-time lower bound *the reduction limitation*.

From the discussion above, we have,

Theorem 3: Both the DMM and the UMM with width w, and latency l takes at least $\Omega(\frac{n}{w} + l \log n)$ time units to compute the sum of n numbers.

From Theorem 3, Algorithm Simple for Lemma 2 is optimal.

V. A SUMMING ALGORITHM USING ZERO BARRIER SYNCHRONIZATION STEP

This section section shows a summing algorithm using zero barrier synchronization step.

Clearly, if we use a single warp of w threads, then no barrier synchronization is necessary. Let us consider that the input is given in a a-dimensional array a of size $\frac{n}{w} \times w$ ($\frac{n}{w}$ rows and w columns). First, the sum of each column is computed using a thread. After that, the sum of the column-wise sum is computed using Algorithm Simple (Lemma 2). The details of the algorithm are spelled out as follows:

[Algorithm One-Warp]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $t \leftarrow 1$ to $\frac{n}{w} - 1$ do
  T(i) performs $a[0][i] \leftarrow a[0][i] + a[t][i]$
Compute $a[0][0] + a[0][1] + \cdots + a[0][w-1]$
 using Algorithm Simple.

The computation of the column-wise sum performs contiguous access. Thus, from Lemma 1, it takes $O(\frac{nl}{w})$ time. After that, Algorithm Simple computes the sum of w numbers in $O(l \log w)$ time. Thus, we have,

Lemma 4: Algorithm One-warp computes the sum of n numbers in $O(\frac{n}{w}l + l \log n)$ time units and 0 barrier synchronization step using w threads on the DMM and on the UMM with width w and latency l.

Clearly, the computing time has an overhead of factor l, and hence Algorithm One-warp is not time optimal.

VI. A SUMMING ALGORITHM BASED ON A $2w$-ARY TREE

We need to use more than w threads to obtain a time-optimal summing algorithm. However, if we use more than w threads, barrier synchronization is necessary. This section shows a summing algorithm using more than w threads. The goal of the summing algorithm shown in this section is to minimize the number of barrier synchronization steps.

For simplicity, we assume that $n = (2w)^k$ for some integer k. We can build $2w$-ary tree with n leaves, each of which corresponds to an input number. The n leaves are partitioned into $\frac{n}{2w}$ groups and each group is connected to an first-level internal node. Thus, we have $\frac{n}{2w}$ first-level internal nodes. The first-level internal nodes are partitioned into $\frac{n}{(2w)^2}$ groups and each group is connected to an second-level internal node. Continuing similarly, we can build a $2w$-ary tree with k-levels.

The computation of the sum is performed from leaves to the root. The sum of each group of the leaves is computed by a warp. The resulting sum is stored in second-level internal nodes. After that, the sum of each group in the second-level is computed by a warp, and the resulting sum is stored in third-level internal nodes. Continuing similarly, we can obtain the sum.

Let a_0 denote the input array, and $a_1, a_2, \ldots, a_k$ be working space each of which corresponds to internal nodes of the tree. Each a_i ($1 \leq i \leq k$) can store $\frac{n}{(2w)^i}$ numbers. Algorithm Tree computes the resulting sum in $a_k[0]$ as follows:

[Algorithm Tree]
for $t \leftarrow 1$ to k do
 for $i \leftarrow 0$ to $\frac{n}{(2w)^t} - 1$ do in parallel
  begin
   W(i) computes $a_t[i] \leftarrow a_{t-1}[i \cdot 2w] +$
    $a_{t-1}[i \cdot 2w + 1] + \cdots + a_{t-1}[(i+1) \cdot 2w - 1]$
    using Algorithm One-warp.
   syncthreads
  end

Let us evaluate the computing time for each t. First, when $t = k$, one warp is used to compute the sum of $2w$ numbers. From Lemma 4, it takes $O(l \log w)$ time units. When $t = k-1$, w warps with w threads each are used. Since each of the warps accesses one row, the contiguous access is performed $\log w$ times. Thus, from Lemma 1, each contiguous access takes $O(\frac{(2w)^2}{w} + \frac{(2w)^2 l}{2w^2}) = O(w + l)$ time units. Hence, the computing time for $t = k-1$ is $O((w+l) \log w)$. Let us consider the general case for $t = k-j$ ($0 \leq j \leq k-1$). The contiguous access for $(2w)^{j+1}$ numbers is performed by $(2w)^j$ warps of $(2w)^j \cdot w$ threads. Thus, the contiguous access takes $O(\frac{(2w)^{j+1}}{w} + \frac{(2w)^{j+1} l}{(2w)^j \cdot w}) = O((2w)^j + l)$ time units. Since the contiguous access is repeated $O(\log w)$ times, the total computing time for $t = k - j$ is $O(((2w)^j + l) \log w)$. Hence, the total computing time of Algorithm Tree is:

$$\sum_{t=1}^{k} O(((2w)^j + l) \log w)$$

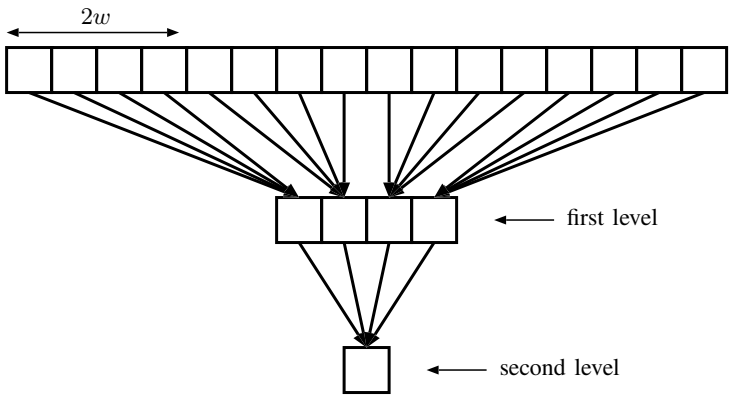

Figure 7. A summing algorithm based on a $2w$-ary tree

$$\begin{aligned} &= O(((2w)^k + kl)\log w) \\ &= O(\frac{n}{w}\log w + l\log n) \quad \text{From } k = \tfrac{\log n}{\log(2w)}. \end{aligned}$$

Also, Algorithm Tree performs *syncthreads* $k = \frac{\log n}{\log(2w)}$ times. Thus, we have,

Lemma 5: The sum of n numbers can be computed in $O(\frac{n}{w}\log w + l\log n)$ time units with $O(\frac{\log n}{\log w})$ barrier synchronization steps using $\frac{n}{2}$ threads on the DMM and on the UMM with width w and latency l.

From Theorem 3, the algorithm for this lemma is not time optimal.

VII. A TIME-OPTIMAL ALGORITHM FOR COMPUTING THE SUM USING FEW BARRIER SYNCHRONIZATION STEPS

We can obtain time optimal summing algorithms with compensation of few additional barrier synchronization steps. This section is devoted to show such time optimal summing algorithms.

Suppose that Algorithm Simple is executed for $t = m-1, m-2, \ldots, m-\log\log w$. It should be clear that the interim sum are stored in $a[0], a[1], \ldots, a[\frac{n}{\log w} - 1]$. After that, the sum of these $\frac{n}{w}$ numbers are computed by Algorithm Tree. The details are spelled out as follows:

[Algorithm Simple-Tree]
for $t \leftarrow m-1$ down to $m - \log\log w$ do
 begin
  for $i \leftarrow 0$ to $2^t - 1$ do in parallel
   T(i) performs $a[i] \leftarrow a[i] + a[i + 2^t]$
  syncthreads
 end
Use Algorithm Tree to compute the sum $a[0] + a[1] + \cdots + a[\frac{n}{\log w} - 1]$.

Let us evaluate the computing time. As we have discussed, Algorithm Simple takes $O(\frac{2^t}{w} + l)$ time units for each t. Thus, the execution of Algorithm Simple for $t = m-1, m-2, \ldots, m - \log\log w$ takes

$$\begin{aligned} \sum_{t=m-\log\log w}^{m-1} O(\frac{2^t}{w} + l) &= O(\frac{2^m}{w} + l\log\log w) \\ &= O(\frac{n}{w} + l\log\log w). \end{aligned}$$

Also, it has $\log\log w$ barrier synchronization steps. After that, Algorithm Tree is executed for the input of size $\frac{n}{\log w}$. From Lemma 5, it takes

$$O(\frac{\frac{n}{\log w}}{w}\log w + l\log\frac{n}{\log w})) = O(\frac{n}{w} + l\log n)$$

time units. Further, it has $O(\frac{\log \frac{n}{\log w}}{\log w}) = O(\frac{\log n}{\log w})$ synchronization steps. Thus, we have,

Theorem 6: Algorithm Simple-Tree computes the sum of n numbers in $O(\frac{n}{w} + l\log n)$ time units with $O(\frac{\log n}{\log w} + \log\log w)$ barrier synchronization steps using $\frac{n}{2}$ threads on the DMM and on the UMM with width w and latency l.

We will show that, the number of syncthreads can be independent of the number n of input numbers. Suppose that input n ($\geq wl$) numbers are stored in an array a of size $\frac{n}{wl} \times wl$ ($\frac{n}{wl}$ rows and wl columns). First, we assign one thread to each row and compute the column-wise sum. After that, the sum of the column-wise sums using Algorithm Simple-Tree. The details are spelled out as follows:

[Algorithm Hybrid]
for $i \leftarrow 0$ to $wl - 1$ do in parallel
 for $t \leftarrow 1$ to $\frac{n}{wl} - 1$ do
  T(i) performs $a[0][i] \leftarrow a[0][i] + a[t][i]$
Use Algorithm Simple-Tree to compute the sum $a[0][0] + a[0][1] + \cdots + a[0][wl-1]$.

Let us evaluate the computing time. The column-wise sum performs contiguous access. Thus, from Lemma 1, it takes $O(\frac{n}{w} + \frac{nl}{wl}) = O(\frac{n}{w})$ time units. After that, Algorithm Tree is executed for wl numbers. From Theorem 6, it

takes $O(\frac{wl}{w} + l\log(wl)) = O(l\log n)$ time units using $O(\frac{\log(wl)}{\log w} + \log\log w) = O(\frac{\log l}{\log w} + \log\log w)$ barrier synchronization steps. Thus, we have,

Theorem 7: Algorithm Hybrid computes the sum of n numbers in $O(\frac{n}{w} + l\log n)$ time units with $O(\frac{\log l}{\log w} + \log\log w)$ barrier synchronization steps using wl threads on the DMM and on the UMM with width w and latency l.

VIII. Conclusion

The main contribution of this paper is to introduce the asynchronous version of the memory machine models, the DMM and the UMM. We also presented time-optimal parallel summing algorithm running in $O(\frac{n}{w} + l\log n)$ time units and $O(\frac{\log l}{\log w} + \log\log w)$ barrier synchronization steps. It is an interesting open problem to further reduce the number of barrier synchronization steps of time-optimal parallel summing computation.

References

[1] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*. Addison Wesley, 1983.

[2] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[3] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*. Addison Wesley, 2003.

[4] M. J. Quinn, *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

[5] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[6] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using Hough transform on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 313–319.

[7] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

[8] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 153–159.

[9] K. Ogawa, Y. Ito, and K. Nakano, "Efficient canny edge detection using a gpu," in *Proc. of International Conference on Networking and Computing*, Nov. 2010, pp. 279–280.

[10] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 320–326.

[11] ——, "Accelerating the dynamic programming for the optial poygon triangulation on the GPU," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, Sept. 2012, pp. 1–15.

[12] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 4.0," 2011.

[13] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, pp. 260–276, July 2011.

[14] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[15] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2012, pp. 788–797.

[16] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, 1972.

[17] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, Sept. 2012, pp. 99–113.

[18] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The nyu ultracomputer – designing an MIMD shared memory parallel computer," *IEEE Trans. on Computers*, vol. C-32, no. 2, pp. 175 – 189, Feb. 1983.

[19] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. on Computers*, vol. C-24, no. 12, pp. 1145– 1155, Dec. 1975.

[20] S. G. Akl, *Parallel Sorting Algorithms*. Academic Press, 1985.

[21] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, 1968, pp. 307–314.